



**Catedráticos:** Ing. Bayron López, Ing. Edgar Sabán

**Tutores académicos:** Julio Flores, Andrea Alvarez, Jordy González, Kevin Lajpop

# FileVersion-3D

*Segundo proyecto de laboratorio*

## 1. Objetivos

### 1.1 Objetivo General

Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 2 en la creación de soluciones de software.

### 1.2 Objetivos Específicos

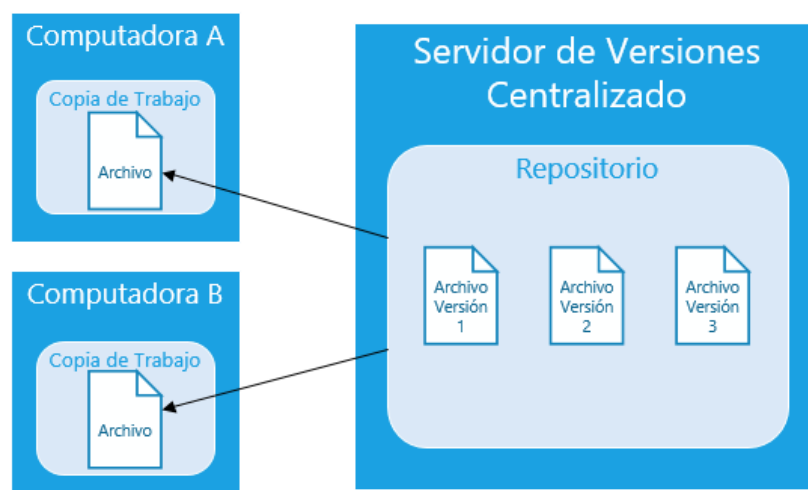
- Aplicar los conceptos de compiladores para implementar el proceso de transformación del código de alto nivel a código intermedio utilizando la representación de tres direcciones.
- Diferenciar el concepto de intérprete y compilador mediante la construcción de un intérprete para código intermedio.
- Aplicar el concepto de optimización de código mediante la implementación de un proceso de optimización de código intermedio utilizando la técnica de optimización por bloques.
- Aplicar los conceptos de compiladores para implementar el proceso de transformación del código intermedio a código de lenguaje ensamblador utilizando la sintaxis de procesadores Intel 386 a manera de poder compilar un archivo ejecutable.

## 2. Descripción del proyecto

Para este proyecto debe crearse un servidor de versiones. Un Servidor de Versiones es una herramienta que ayuda al programador a administrar, almacenar y registrar cambios en el código. Actualmente existen varios servidores de versiones como: Git, Mercurial, Subversion, SourceSafe, entre otros. Existen diferentes tipos de versionamiento: Local, Centralizado o Distribuido. Para el proyecto únicamente debe implementarse un versionamiento Centralizado. Por lo tanto existen 2 términos importantes que deben tomarse en cuenta y se definen a continuación:

- **El Repositorio:**  
Es una base de datos de archivos que contiene todos los archivos cuyas versiones se controlan con sus respectivas historias. El repositorio normalmente yace en un servidor de archivos, que provee a pedido el contenido a los clientes.
- **La Copia de Trabajo:**  
Cada desarrollador tiene su propia copia de trabajo en su computador local, desde la cual puede obtener la última versión del repositorio, trabajar en ella localmente sin modificar el código en el repositorio, y cuando esté satisfecho con los cambios que ha realizado puede confirmar sus cambios en el repositorio.

Figura 1. Diagrama de modelo de versionamiento centralizado



El servidor de Versiones a desarrollarse ha de llamarse FileVersion y debe cumplir con el modelo centralizado. Dicho servidor aloja archivos de código de alto nivel. Por lo tanto, un cliente del servidor FileVersion puede obtener una copia de código del Repositorio y trabajarla en su computadora, y puede subir su código al Repositorio generando una nueva versión.

Además del funcionamiento normal del servidor de versiones, deben implementarse las siguientes características extendidas, que aplican al código de alto nivel que se encuentra en el repositorio (el código de alto nivel a trabajar se define en la sección 4 de este documento):

- El servidor debe ser capaz de generar código intermedio utilizando el código de alto nivel que se encuentra en el repositorio.
- El servidor debe ser capaz de interpretar el código intermedio generado previamente.
- El servidor debe ser capaz de optimizar el código intermedio generado previamente.
- El servidor debe ser capaz de generar el código ensamblador utilizando el código intermedio generado previamente.

Para cumplir con las funcionalidades previamente mencionadas, FileVersion se compone por un conjunto de programas o módulos que cumplen con las diferentes tareas, y que funcionan en conjunto como una sola aplicación. A continuación se describen brevemente estos módulos (en la sección 3 de este documento se encuentra una explicación técnica de su funcionamiento):

- Primer módulo (Cliente FileVersion): este módulo permite la interacción del usuario con la aplicación a través de una interfaz gráfica. La interfaz permite al usuario utilizar una serie de comandos que le indican al servidor de versiones qué acciones realizar sobre el código en el Repositorio o en la Copia de Trabajo. Ante cada uno de los comandos que el usuario ingresa, el servidor debe retornar un mensaje o reporte, por lo tanto éste módulo cliente debe tener la capacidad de mostrar estos mensajes o reportes. Este módulo debe poder ejecutarse varias veces desde distintos lugares para poder ser utilizado por varios usuarios a la vez. Este módulo no debe de modificar directamente ningún archivo. Toda modificación debe ser ajena a la aplicación y realizarse en un editor de texto.

Figura 2. Diagrama de funcionalidades del Cliente FileVersion



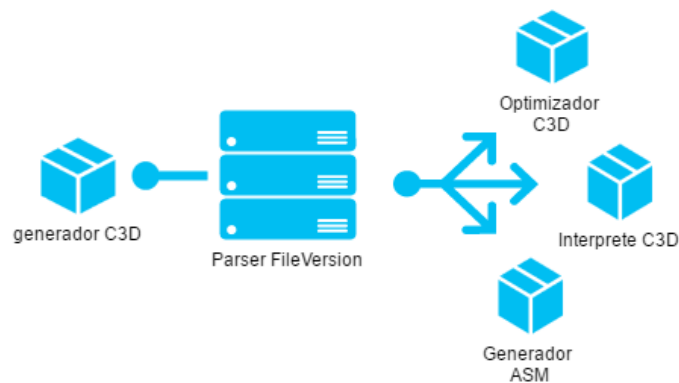
- Segundo módulo (Servidor FileVersion): este módulo es el encargado del versionamiento de los archivos. Dicho módulo puede almacenar, administrar y registrar cambios en los archivos en el repositorio. Por lo tanto este servidor debe mantener un registro de todas las versiones que han existido para cada archivo. Este módulo tiene una página web como interfaz de acceso, que permite visualizar los archivos que se encuentran en el repositorio.

Figura 3. Diagrama de funcionalidades del Servidor FileVersion



- Tercer módulo (Parser FileVersion): Este módulo es el encargado de realizar el análisis del lenguaje de alto nivel orientado a objetos e implementar las funcionalidades extra al servidor de versiones. Tendrá la capacidad de traducir las instrucciones del lenguaje de alto nivel a código intermedio. También deberá contar con un optimizador de código de tres direcciones, con la finalidad de optimizar el código intermedio generado y obtener un código intermedio más eficiente. Además debe poder interpretar el código intermedio generado para validar la correcta funcionalidad del mismo. Finalmente debe de poder traducir el código intermedio generado para producir código de lenguaje ensamblador, que podrá ser compilado del lado del cliente.

Figura 4. Diagrama de funcionalidades del Parser FileVersion



Estos tres módulos se comunican por medio de Apache Thrift para enviar y recibir solicitudes, y funcionar como una sola aplicación.

Figura 5. Diagrama de componentes del Cliente FileVersion



## 3. Funcionalidad de Componentes de la Aplicación

La funcionalidad de la aplicación está dividida en tres módulos; cliente, servidor y analizador (parser), los módulos se comunican entre sí, enviando y recibiendo archivos que logran todo el proceso del versionamiento y ejecución de código PHP, a continuación se describe a detalle la funcionalidad de cada uno de estos módulos y la forma en que interactúan entre sí.

### 3.1 Cliente FileVersion

Cliente FileVersion, corresponde al módulo de la aplicación que interactúa con el usuario para la comunicación con el servidor de versiones, este módulo será realizado en lenguaje C#. Para comprobar el funcionamiento de este módulo, se deberá crear una o dos máquinas virtuales con plataforma windows, las cuales simularán los clientes y en ellas se ejecutara este módulo.

Al momento de ejecutar el Cliente FileVersion se deberá generar una carpeta en C:/ con el nombre "FileVersionClient", esta carpeta funcionará como una copia de trabajo, en esta carpeta se alojarán todos los proyectos que se quieren versionar y/o ejecutar, dichos proyectos contendrán archivos con lenguaje alto nivel orientado a objetos (ver sección 4 de este documento para descripción del lenguaje).

### 3.1.1 Comandos de Instrucciones FileVersion

El cliente FileVersion cuenta con una serie de comandos que le dan instrucciones al servidor, y de esta manera realizar acciones sobre los proyectos alojados en el Repositorio o en la Copia de Trabajo. Cada comando tiene una funcionalidad distinta para poder realizar la administración y el versionamiento de los archivos, así como la ejecución de dichos archivos. Para aplicar las distintas acciones a los proyectos se utilizarán los siguientes comandos:

- **Source init “nombre del proyecto”**  
Comando que crea un repositorio de manera física en el servidor de versiones y en la máquina donde se esté ejecutando la aplicación Cliente FileVersion (carpeta de la copia de trabajo). Esta acción generará una llave única que debe ser numérica, dicha llave sirve para identificar al repositorio y será generada automáticamente por el servidor de versiones.
- **Source -d “llave única”**  
Comando que elimina el repositorio de manera física en el servidor de versiones. Con esta acción se desentrelaza los repositorios, dejando solo los archivos en la copia de trabajo.
- **Source -s “nombre del proyecto” “llave única”**  
Comando que cambia de repositorio, por ejemplo, si se está trabajando en un repositorio en específico (“proyecto x”) y se desea cambiar a otro repositorio (“proyecto y”) se deberá utilizar este comando. A continuación se describe cada parámetro del comando.
  - El parámetro -s es para hacer un cambio (“switch”) de proyecto.
  - El parámetro “nombre del proyecto”, selecciona la carpeta de la copia de trabajo. Si no existe el proyecto en la copia de trabajo debe de desplegar el error en la sección de resultados de la interfaz.
  - El parámetro “llave única” selecciona la carpeta del repositorio a la que pertenece dicha llave única. En dado caso no exista la llave única el servidor de versiones debe comunicar que no existe un repositorio con esa llave y debe de desplegar el error en la sección de resultados de la interfaz.

Nota: Los comandos que se ejecuten después, afectarán al repositorio que se seleccionó con esta acción.

- **Source commit [.|+] “nombre archivo”**  
Comando que sube archivos contenidos en la carpeta de la copia de trabajo al repositorio. A continuación se describe cada parámetro del comando:
  - El parámetro [.|], subirá **todos** los archivos que están en la copia de trabajo al repositorio.
  - El parámetro[.+], subirá **solo** archivos que se modificaron en la copia de trabajo.
  - El parámetro “nombre archivo”, subirá solo un archivo en específico.

Nota: Esta acción generará una “llave de commit” la cual servirá para regresar a una versión anterior del repositorio.

- **Source update**

Comando que descarga la última versión de todos los archivos que están en el repositorio y los copiará en la carpeta de la copia de trabajo.

- **Source update -r “llave de commit”**

Comando que restaura la versión correspondiente a la clave del commit de todos los archivos que están en el repositorio y los moverá a la carpeta de la copia de trabajo.

- **Prueba “llave única” -r “llave de commit”**



Comando que interpreta el código (PHP) que está en el repositorio identificado con la llave única dada y la versión del código correspondiente a la llave del commit, los resultados se debe mostrar en la sección de resultados de la interfaz.

Nota: la ejecución del código se debe de realizar por medio del compilador de php, con un comando Shell.

- **Prueba -3D “llave única”**

Comando que genera e interpreta el código 3 direcciones de la última versión de los archivos que se encuentra en el repositorio correspondientes a la llave única dada y mostrará en la aplicación el resultado de lo interpretado y la tabla de símbolos que se utilizó para generar el código 3 direcciones.

- **Release -3D -o “llave única”**

Comando que genera el código 3 direcciones optimizado (por medio del método de bloques) de la última versión de los archivos que se encuentra en el repositorio correspondiente a la llave única dada y mostrará en la aplicación el código 3 direcciones optimizado generado y la tabla de símbolos que se utilizó para generarlo.

- **Release -3D “llave única”**

Comando que genera únicamente el código 3 direcciones normal de la última versión de los archivos que se encuentra en el repositorio correspondiente a la llave única dada y mostrará en la aplicación el código 3 direcciones generado con la tabla de símbolos que se utilizó para generarlo.

- **Release -asm “llave única”**

Comando que genera el código 3 direcciones de la última versión de los archivos que se encuentra en el repositorio correspondientes a la llave única dada, para luego convertirlo en su equivalente en código assembler y mostrará en la aplicación el código assembler generado.

- **Release “llave única”**

Comando que genera el ejecutable (.exe) correspondiente al código (PHP) de la llave única dada, para esto primero genera el código 3 direcciones que se encuentra en el repositorio, luego lo convierte a lenguaje ensamblador, por último compilar el código ensamblador (ver sección 7 de este documento).

### 3.1.2 Interfaz Cliente FileVersion

La interfaz del Cliente FileVersion, debe contar con un área de entrada, la cual permitirá ingresar los comandos de instrucciones. Esta área debe manejar múltiples pestañas para que sea más eficiente y ordenado el manejo de las copias de trabajo, también debe contar con un área de reportes y/o salidas de las respuestas del repositorio, en dicha área se mostrará la tabla de símbolos utilizada para generar código 3 direcciones de los archivos (PHP) en formato html y los reportes de errores léxicos, semánticos y sintácticos de los archivos (PHP) analizados (ver sección 8 de este documento) en formato html, ambos (tabla de símbolos y reportes de errores) deberá ser cargado a un visor html dentro de la aplicación, y los resultados que se generan deberán estar en formato txt.

Nota: al hacer uso de cada comando se debe desplegar en el área de resultados si la acción ha sido exitosa o no, y también mostrar el resultado si se ha solicitado alguna ejecución de los archivos.

Figura 6. Interfaz sugerida para Cliente FileVersion

▼ Ámbito	▼ Acceso	▼ Nombre	▼ Tipo	▼ Posición	▼ Tamaño	▼ Dimensión



## Ejemplo del funcionamiento de Cliente FileVersion

A continuación se muestra un ejemplo del funcionamiento total esperado por parte de la interfaz del cliente FileVersion.

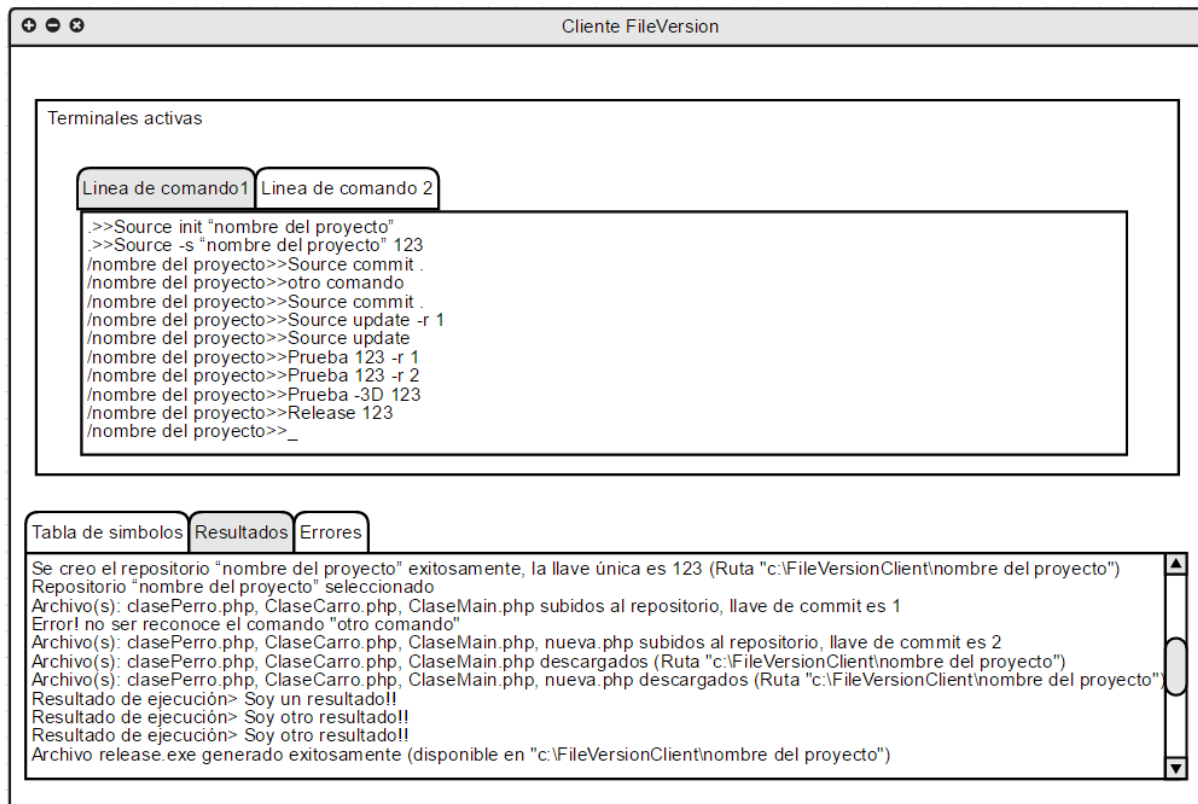


Figura 7. Interfaz funcionando del Cliente FileVersion

- Para este ejemplo se ingresaron 11 comandos, para cada uno de los comandos ingresados se obtuvo 11 respuestas que se muestran en la sección de resultados de la interfaz sugerida.
- Se realizaron 2 commits en los cuales antes del segundo commit se creó de manera física el archivo nueva.php en la copia de trabajo.
- Cuando se ejecuto el comando "prueba" se realizó en base a los commits realizados por lo que los resultados de la ejecución son diferentes, ya que se modificó el código fuente en el segundo commit.

## 3.2 Servidor FileVersion

Servidor FileVersion, es el módulo administrador de todas las acciones que el Cliente FileVersion solicite. El servidor de gestión de versiones será realizado en lenguaje Java. En dicho servidor se almacenarán todos los repositorios de código de los clientes, se sugiere que al momento de ejecutar el Servidor FileVersion se genere una carpeta en C:/ con el nombre "FileVersionServer", esta carpeta funciona como el Repositorio de la aplicación. Los clientes podrán acceder al Repositorio por medio de los comandos descritos previamente en Cliente FileVersion, y también se podrá acceder a dicha carpeta por medio de una página web, que funciona como interfaz gráfica del servidor.


En la Página Web se debe mostrar la lista de los proyectos existentes. Al seleccionar uno de los proyectos, se mostrará la estructura en la que están almacenados los archivos del mismo y debe poder visualizarse el contenido de cada archivo seleccionada una revisión (llave de commit).

Esta aplicación contará con un registro de repositorios creados por los clientes, el cual ayudará a recordar cuántos repositorios posee (llaves únicas) y cuantas commits (llaves de commit) hay para cada repositorio además de registrar las rutas de físicas de las mismas en un archivo. Este archivo será de importancia ya que con él se atenderán las solicitudes del cliente y la página web (el formato y la administración de dicho registro queda a discreción del estudiante).

Las herramientas utilizadas para la creación de la página web quedan a discreción del estudiante.

Figura 8. Interfaz sugerida para Servidor FileVersion

### FileVersion



The screenshot shows a web interface for 'FileVersion'. At the top, there is a header bar with the title 'Proyectos'. Below this, there is a list of three items: 'Programa 1', 'Proyecto', and 'Calculadora'. Each item is displayed in a separate row with a light blue background and a thin border. The text is in a sans-serif font.

Proyectos
Programa 1
Proyecto
Calculadora

Figura 9. Estructura del Proyecto seleccionado (esta estructura es correspondiente a la revisión seleccionada)

## FileVersion

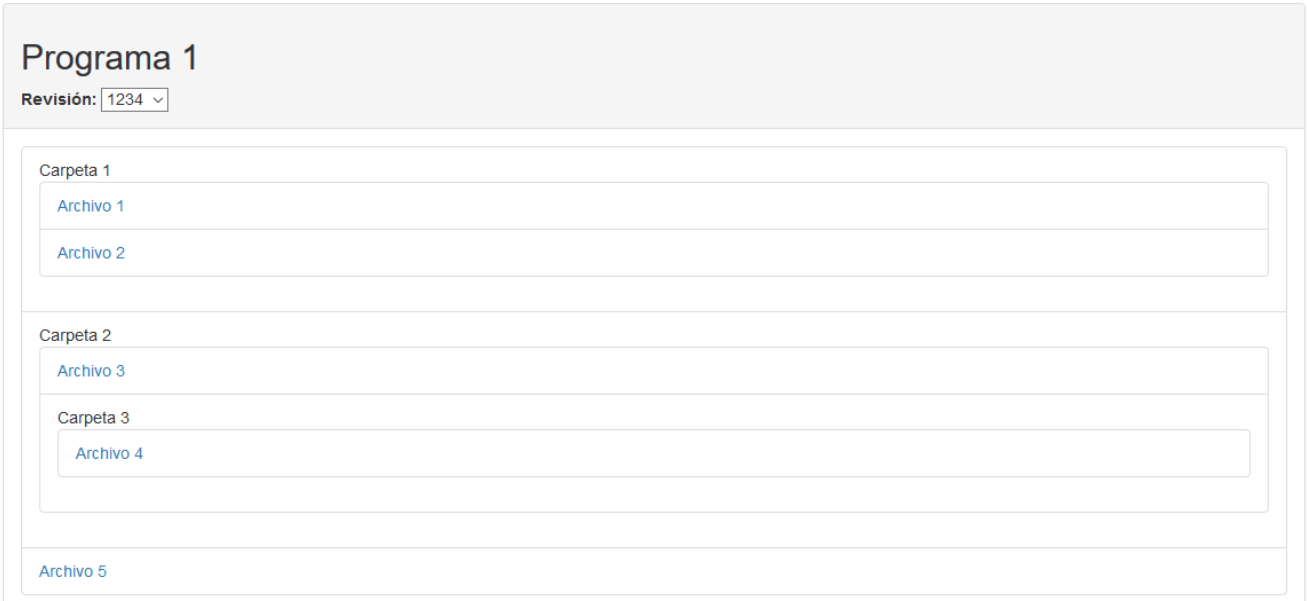
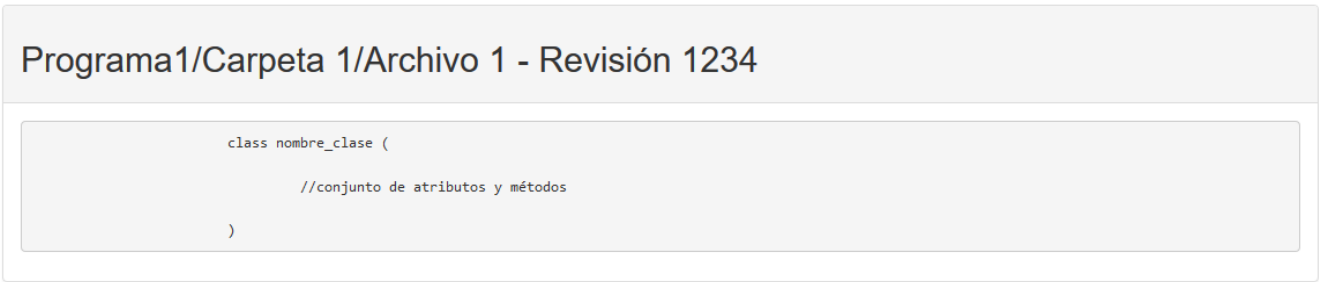


Figura 10. Vista de contenido del archivo seleccionado

## FileVersion



## 3.3 Servidor Parser FileVersion

El servidor Parser FileVersion, es un módulo que utiliza un conjunto de 4 analizadores que realizan las funcionalidades extendidas de la aplicación y que permiten responder a las peticiones del cliente. El servidor Parser FileVersion debe realizarse en lenguaje Java. A continuación se describen los analizadores mencionados:

### 3.3.1 Generador de código intermedio

La función principal del Generador de código Intermedio es la generación del código 3 direcciones asociado a un código de entrada php. Este código debe generarse de acuerdo al estándar de código intermedio visto en clase (Para ver el estándar de código intermedio consultar sección 5 de este documento). Se utilizará JavaCC para el análisis de los archivos con lenguaje PHP. En caso de existir un error léxico, sintáctico o semántico en el código de entrada debe enviarse un reporte de errores al cliente (ver sección 8 de este documento), el cual lo mostrará al usuario.

Al finalizar la generación de código, este debe retornar al servidor de versiones, en donde se almacenará en una carpeta especial (asociada a la revisión o identificador de commit). Esta carpeta se llamará: **\_\_COD3DIR**

Como cualquier carpeta del proyecto esta debe ser visible desde la página web del servidor de versiones. Además también debe enviarse una respuesta a la aplicación cliente con un mensaje de éxito o fracaso y en caso de éxito, la dirección donde puede visualizarse el código desde la página web.

Figura 11. Diagrama entradas y salidas del generador de código intermedio



### 3.3.2 Intérprete de código intermedio

El intérprete de código intermedio permite la ejecución de las instrucciones del código 3 direcciones generado. El intérprete debe ser capaz de llevar a cabo todas las instrucciones, incluidas aquellas que generen una salida. En caso de existir una salida deberá mostrarse el resultado de la ejecución en la aplicación cliente. Para ello, el módulo analizador debe ser capaz de comunicarse con el servidor de versiones y enviarle la cadena de salida. El módulo de Servidor de versiones por su parte debe reenviar la salida al cliente que solicitó la prueba. Se utilizará JavaCC para la interpretación.

Figura 12. Diagrama entradas y salidas del intérprete de código intermedio



### 3.3.3 Optimizador de código intermedio

El Optimizador de código Intermedio permitirá aplicar las reglas del método de optimización por Bloques (Para ver reglas de optimización por bloques consultar sección 6 de este documento), a manera que el código obtenido sea más eficiente que el código generado. El optimizador no ejecutará código, simplemente lo optimizará y podrá guardar el código resultante en un archivo con extensión .c3do en la carpeta previamente creada **\_\_COD3DIR**, adicionalmente se deberá generar y guardar en la misma carpeta, un archivo log con las operaciones que realizó el método de bloques, dicho archivo tendrá un formato html (ver figura 14 para referencia de formato). El optimizador se desarrollará en el lenguaje de programación Java y se apoyará en la herramienta JavaCC para realizar el análisis léxico, sintáctico y semántico del código de tres direcciones a optimizar.

Figura 13. Diagrama entradas y salidas del optimizador de código intermedio

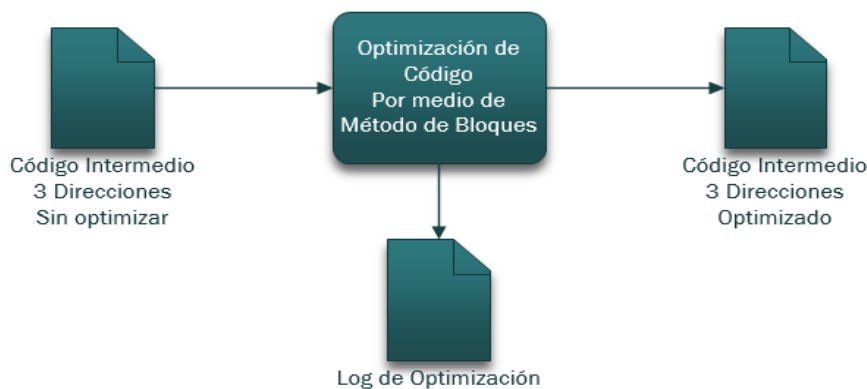


Figura 14. Interfaz del formato sugerido para log de optimización por bloques

Caso	Tipo	Temporal afectado	Nuevo valor	Operacion
Sub Expresiones Comunes Locales	Cambio	t6	t4	p + 2
Codigo inactivo	Eliminacion	t8		
Propagacion de copias	Eliminacion de referencias	t8		
Sub Expresiones Comunes Locales	Cambio	t11	t9	p + 2
Codigo inactivo	Eliminacion	t13		
Propagacion de copias	Eliminacion de referencias	t12		

### 3.3.4 Traductor a código ensamblador

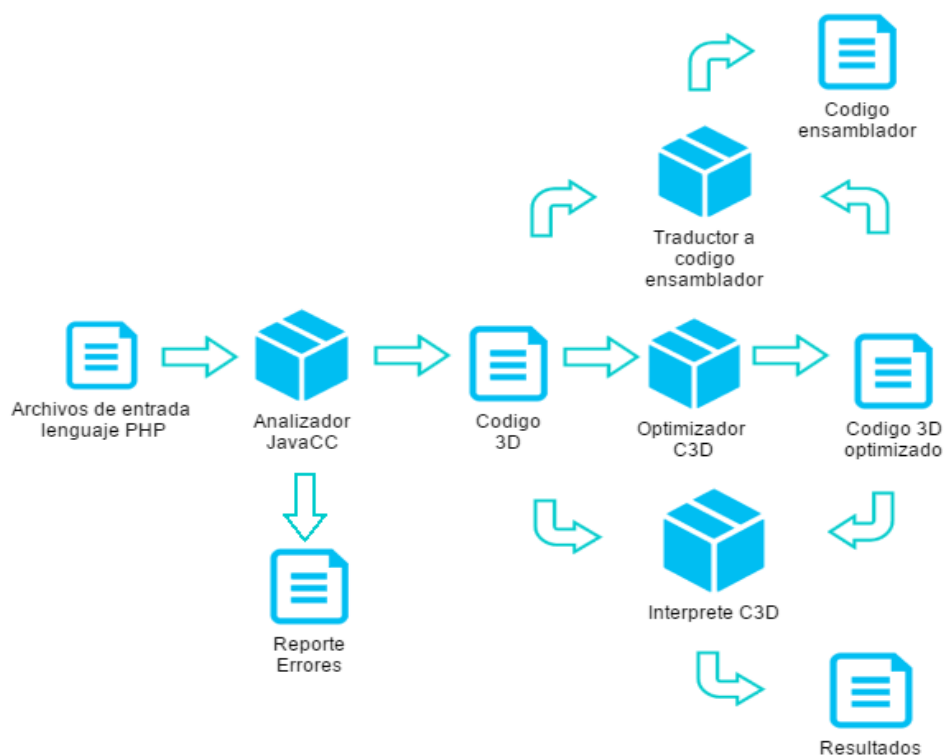
Consiste en la traducción del código intermedio, optimizado o no, a código ensamblador. Para realizar esta acción el estudiante deberá escribir una definición dirigida por la sintaxis que reconozca la sintaxis definida posteriormente para el código de tres direcciones y lo traduzca al formato de código ensamblador (Para ver el estándar de código ensamblador consultar sección 7 de este documento), para poder generar código objeto. El código objeto, debe ser generado utilizando la herramienta NASM y compilarlo del lado del cliente FileVersion, usando el compilador GCC y de esta manera crear un ejecutable, esto significa que el código ensamblador resultante debe poder ejecutarse en plataforma Windows. El código ensamblador generado debe almacenarse en una carpeta especial del repositorio asociada a la revisión. Esta tendrá por nombre: **\_\_CODASM**

Además debe enviarse una respuesta al cliente con el código (para su posterior compilación).

Figura 15. Diagrama entradas y salidas del traductor de código ensamblador



Figura 16. Diagrama de Componentes completo del módulo Parser FileVersion



## 4. Descripción del lenguaje orientado a objetos *PHP*

Toda la funcionalidad de FileVersion (sección 3 de este documento) gira entorno a archivos que contienen un lenguaje de alto nivel, el lenguaje a utilizar será php, limitándose a una funcionalidad básica de lenguaje que de manera pueda soportar funcionamiento básico con programación orientada a objetos. Las funcionalidades que se requieren son las siguientes:

- **Comentarios**

Los comentarios que maneja php son de una línea y de varias líneas:

```
<?php

//comentario de una línea

/* comentario de
varias
líneas
*/
?>
```

- **Variables locales y globales**

Dentro de un entorno global las variables (atributos) se definen de la siguiente manera:

```
<?php

class clase1 {

var $soynumero, $soynumero2=1;
var $caracter='a';
var $soycadena = "hola mundo";
var $soydecimal = 1.2;

}

?>
```

Las variables locales se definen similar con la diferencia de que no se le antepone la sentencia "var".

```
$soynumero_local;
$soynumero_local = 10;
```

- **Vectores y matrices**

Php tienen muchas formas de manejar arreglos y matrices, pero a la vez carece de una sintaxis en específico, para términos del proyecto se utilizará la sintaxis convencional como se puede ver a continuación:

```
var $a;  
  
function matriz(){  
    for($i=0;$i<10; $i++){  
        for($j=0; $j<10; $j++){  
            $a[$i][$j]= $i*$j;  
        }  
    }  
  
    $a[10][10] = 100;  
    echo $a[10][10];  
}
```

Se declara de la misma manera que cualquier variable, con la diferencia es cuando se le asigna un valor en forma matricial o vectorial, el compilador lo toma como una matriz o vector.

Un vector es una matriz unidimensional y es tratada de la misma manera, por ejemplo:

```
var $a;  
  
function vector(){  
    for($j=0; $j<10; $j++){  
        $a[$i]= $i;  
    }  
  
    echo $a[9];  
}
```



- **Operaciones aritméticas**

Las operaciones aritméticas aceptadas por php son las siguientes:

Operadores aritméticos		
Ejemplo	Nombre	Resultado
$- \$a$	Negación	Opuesto de $\$a$ .
$\$a + \$b$	Adición	Suma de $\$a$ y $\$b$ .
$\$a - \$b$	Sustracción	Diferencia de $\$a$ y $\$b$ .
$\$a * \$b$	Multipliación	Producto de $\$a$ y $\$b$ .
$\$a / \$b$	División	Cociente de $\$a$ y $\$b$ .
$\$a \% \$b$	Módulo	Resto de $\$a$ dividido por $\$b$ .
$\$a ** \$b$	Exponenciación	Resultado de elevar $\$a$ a la potencia $\$b$ ésima. Introducido en PHP 5.6.

Tomado de la documentación oficial de php: <http://php.net/manual/es/language.operators.arithmetic.php>

- **Operaciones relacionales**

Php se distingue de manejar un gran conjunto de operaciones relacionales, pero para el presente proyecto se trabajarán las siguientes:

Operadores de comparación		
Ejemplo	Nombre	Resultado
$\$a == \$b$	Igual	<b>TRUE</b> si $\$a$ es igual a $\$b$ después de la manipulación de tipos.
$\$a === \$b$	Idéntico	<b>TRUE</b> si $\$a$ es igual a $\$b$ , y son del mismo tipo.
$\$a != \$b$	Diferente	<b>TRUE</b> si $\$a$ no es igual a $\$b$ después de la manipulación de tipos.
$\$a <> \$b$	Diferente	<b>TRUE</b> si $\$a$ no es igual a $\$b$ después de la manipulación de tipos.
$\$a !== \$b$	No idéntico	<b>TRUE</b> si $\$a$ no es igual a $\$b$ , o si no son del mismo tipo.
$\$a < \$b$	Menor que	<b>TRUE</b> si $\$a$ es estrictamente menor que $\$b$ .
$\$a > \$b$	Mayor que	<b>TRUE</b> si $\$a$ es estrictamente mayor que $\$b$ .
$\$a <= \$b$	Menor o igual que	<b>TRUE</b> si $\$a$ es menor o igual que $\$b$ .
$\$a >= \$b$	Mayor o igual que	<b>TRUE</b> si $\$a$ es mayor o igual que $\$b$ .

Tomado de la documentación oficial de php: <http://php.net/manual/es/language.operators.comparison.php>

- **Operaciones lógicas**

Los operadores lógicos que maneja php son AND, OR, XOR y NOT y de la siguiente manera es que los maneja:

Operadores lógicos		
Ejemplo	Nombre	Resultado
$\$a \text{ and } \$b$	And (y)	<b>TRUE</b> si tanto $\$a$ como $\$b$ son <b>TRUE</b> .
$\$a \text{ or } \$b$	Or (o inclusivo)	<b>TRUE</b> si cualquiera de $\$a$ o $\$b$ es <b>TRUE</b> .
$\$a \text{ xor } \$b$	Xor (o exclusivo)	<b>TRUE</b> si $\$a$ o $\$b$ es <b>TRUE</b> , pero no ambos.
$! \$a$	Not (no)	<b>TRUE</b> si $\$a$ no es <b>TRUE</b> .
$\$a \&\& \$b$	And (y)	<b>TRUE</b> si tanto $\$a$ como $\$b$ son <b>TRUE</b> .
$\$a \ \  \$b$	Or (o inclusivo)	<b>TRUE</b> si cualquiera de $\$a$ o $\$b$ es <b>TRUE</b> .

Tomado de la documentación oficial de php: <http://php.net/manual/es/language.operators.logical.php>

Ejemplo:

```
var $a;  
function logico(){  
  
    $b;  
    $b = 10;  
    if($a == 0 AND $a < $b OR $b == 10 ){  
        echo "entro!";  
    }  
  
}
```

En la sección de sentencias de flujo y bucles también se ejemplifica los operadores lógicos y relacionales.

- **Métodos**

Los métodos en php pueden o no traer parámetros y se definen de la siguiente manera:

```
function soy_metodo(){  
    //aquí van sentencias  
}  
  
function soy_metodo2($parametro1, $parametro2){  
    //aquí van sentencias  
}
```

- **Funciones**

En el caso de las funciones, son similares a los métodos con la salvedad de que deben de tener la sentencia “return”. Una función puede o no traer parámetros.

```
function soy_funcion(){  
    $sum = 10 + 1;  
    return $sum;  
}  
  
function soy_funcion2($parametro1, $parametro2){  
    $sum = $parametro1 + $parametro2;  
    return $sum;  
}
```

- **Parámetros**

Para los métodos y funciones se utilizarán dos tipos de parámetros, por referencia y por valor.

- **Por referencia**

Se envía la posición del parámetro al método o función para que este pueda modificar su valor, únicamente es aceptado el envío de variables y objetos por referencia. Php lo maneja declarando el parámetro con el prefijo "&" de la siguiente manera:

```
function por_referencia(&$referencia1, &$referencia2){  
    $referencia1 = 10;  
    $referencia2 = $referencia2. " mundo";  
}
```

La manera de pasar un parámetro por referencia es de la siguiente manera:

```
$a;  
$b = "hola";  
$ob = new clase2();  
$ob->por_referencia($a, $b);  
echo $a . " \n" . $b . "\n";
```

La salida de las anteriores sentencias es la siguiente:

```
root@compi2:/var/www/html# php clase2.php  
10  
hola mundo  
root@compi2:/var/www/html#
```

El valor 10 se le asignó a la variable "a" dentro del método y persiste fuera de porque fue por referencia, así como la concatenación de la cadena " mundo" a la variable "b" persiste fuera del método porque fue por referencia.

- **Por valor**

Se envía una copia del valor del parámetro para que el método o función pueda utilizarlo, únicamente es aceptado el envío de variables, valores puntuales y funciones. Php lo maneja de la siguiente manera:

```
function por_valor($valor1, $valor2){  
    $valor1 = 10;  
    $valor2 = $valor2. " mundo";  
}
```

La manera de pasar un parámetro por valor es igual al paso de parámetros por referencia, siguiendo con el ejemplo:

```
$a;
$b = "hola";
$obj = new clase2();
$obj->por_valor($a, $b);
echo $a . " \n" . $b . "\n";
```

La salida de las anteriores sentencias es la siguiente:

```
root@comp12:/var/www/html# php clase2.php
PHP Notice: Undefined variable: a in /var/www/html/clase2.php on line 17
PHP Notice: Undefined variable: a in /var/www/html/clase2.php on line 18

hola
root@comp12:/var/www/html#
```

La lógica es la misma que el ejemplo de paso por referencia pero los resultados son distintos, dado que las modificaciones solamente se verán visibles en el ámbito del método “por\_valor”, fuera de no tiene incidencia, por eso php despliega que la variable “a” se está utilizando pero no ha sido iniciada.

- **Clases**

```
<?php

class clase2{

//sentencias
}

?>
```

Para crear una clase en php se hace de la siguiente manera:

- Constructor  
El constructor en php es un método que se define con el nombre \_\_construct y puede o no recibir parámetros.

```
<?php

class clase2{

function __construct(){
echo "constructor :D";
}

}

?>
```

- Objetos e Importaciones (include)

Para crear un objeto de una clase que no se encuentre en el archivo se debe de importar el archivo con la función “include”.

Para crear un objeto se realiza mediante la sentencia “new”, un objeto puede ser local o global como se observa en el ejemplo siguiente:

```
<?php
include ('prueba.php');

class clase2{
var $obj2;

function __construct(){
echo "constructor :D";

$obj1 = new clase1();
$obj2 = new clase1();
}

}
?>
```

- Herencia

Para poder heredar de una clase a otra php utiliza la sentencia extends de la siguiente manera:

```
<?php
include "clase1.php";

class clase2 extends clase1{

}

?>
```

- Sentencia parent

La sentencia parent se utiliza para acceder a una constante o función de la clase padre, por ejemplo:

```
parent::__construct();//llamada al constructor del padre
parent::holamundo(); //llamda al metodo holamundo del padre
```

- Sentencias this

La sentencia this se utiliza para referenciar al objeto actual y poder acceder a sus atributos.

```
var $a;  
function __construct(){  
    parent::__construct();//llamada al constructor del padre  
    parent::holamundo(); //llamda al metodo holamundo del padre  
    $this->a = 0;  
}  
}
```

- Sentencias de flujo

- If, If else, If else if

La estructura de las sentencias if en php es la siguiente:

```
$a=10;  
$b=15;  
  
//IF-ELSE  
if($a>$b and !($b==15)){  
    $a=0;  
}else{  
    $a = $b;  
}  
  
//IF-ELSEIF  
if($a>$b and !($b==15)){  
    $a=0;  
}else if($b == 15 or $a < 20){  
    $a = $b;  
}else{  
    $b = 0;  
}
```

- Switch

La estructura de la sentencia switch en php es la siguiente:

```
function soy_metodo2($parametro1, $parametro2){
switch($parametro){
case 1:
echo "hola mundo";
break;
case 2:
echo "adios mundo";
break;
case 3:
$parametro1 = 0;
case 4:
$parametro2 = 0;
default:
echo "fin";
}
}
```

lo que evalúa el switch es una expresión como tal. En cada caso puede o no venir la sentencia “break”, si en dado caso viene no ejecuta los demás casos, si no viene sigue ejecutando los demás casos.

- **Bucles**

- While

El ciclo while es un bucle que ejecutará las sentencias asociadas hasta que la condición sea falsa. Su estructura en php es la siguiente:

```
$b= 0;
while(!($b==10) and $b<20){
echo $b;
$b = $b+3;
}
```

- Do while

El ciclo do-while es un bucle que ejecutará primero las sentencias, luego evaluará la condición, y se realiza hasta que la condición sea falsa. Su estructura en php es la siguiente:

```
$b= 0;
do{
$a++;
echo $b;
}while($b == 0 xor $a == 1);
```

- For

El ciclo for es un ciclo que realiza las sentencias un número de veces en específico, teniendo un valor inicial, una evaluación de la condición y un aumento, tal y como se mira en el siguiente ejemplo:

```
for($i=0; $i< 10; $i++){
    $a = $i * 10;
}
```

- Sentencia break y continue

Esta sentencia indicará cuándo debe terminar la ejecución del flujo dentro de las condiciones establecidas puede venir dentro de cualquier estructura de control.

Por ejemplo:

```
for($i=0; $i<10; $i++){
    if($i==5){
        break;
    }
    echo $i;
}
```

La sentencia continue se utiliza únicamente dentro de los bucles para saltar el resto de la iteración actual del bucle y continuar la ejecución en la evaluación de la condición, para luego comenzar la siguiente iteración.

Por ejemplo:

```
while($a < 5){
    $a++;
    if($a<3){
        continue;
    }
    echo $a;
}
```

- **Funciones reservadas**

- echo

Es una función que muestra cadenas (si se envía un valor que no es cadena se convierte a cadena implícitamente).

Ejemplo:

```
$a = 10;
$b = "compi 2";
$c = 1.2;

echo $a . "\n";
echo $b . "\n";
echo $c . "\n";
echo $a . $b . $c . "\n";
```



Las salidas de estas funciones echo son las siguientes:

```
root@compi2:/var/www/html# php inicio.php
10
compi 2
1.2
10compi 21.2
root@compi2:/var/www/html#
```

- **Método “main”**

Todo lenguaje orientado a objetos debe de tener un método dónde inicializar o una forma de inicializar el programa, php carece de método main, entonces para simular un método main se hará lo siguiente:

- Cuando un archivo .php no contiene clases lo que realiza php es ejecutar las sentencias que contiene este archivo, aprovechando esta funcionalidad de php se tendrá un archivo con extensión php que lo único que tendrá será la inicialización del programa (obviamente este archivo no contendrá funcionalidad compleja mucho menos clases), por ejemplo:

```
<?php
include ('prueba2.php');
$obj = new clase2();
?>
```

Como podemos observar este archivo no contiene clases, ni métodos o funciones, únicamente contiene la importación de archivos que contienen las clases y nos servirán para inicializar los objetos, el comportamiento es idéntico a cualquier método estático main.

Nota: tomar en cuenta que toda la descripción anterior fue solamente con fines de ejemplo, se debe de tomar en cuenta la documentación oficial de php <http://php.net/manual/es/langref.php> en el cual se ejemplifican variantes de las sentencias ya descritas, estas sentencias son básicas para el funcionamiento de un programa en php con el paradigma orientado a objetos.

## 5. Descripción de código de 3 direcciones

Cuando el compilador termine la fase de análisis de un programa escrito en php, lo que realizará es una transformación a una representación intermedia equivalente al código de alto nivel (php), esta representación intermedia será el código de tres direcciones. La traducción de las instrucciones de alto nivel se realizará según lo aprendido por el estudiante basado en los conceptos expuestos tanto en clase como en laboratorio, en esta sección se presentará la sintaxis para el código generado.

### 5.1 Estándar de 3 direcciones

El código de tres direcciones seguirá el estándar que se maneja en la clase magistral y de igual manera el que se maneja en el libro de texto.

#### 5.1.1 Temporales

Los temporales se crearán respetando la siguiente sintaxis:

*t<num>*

tomando en cuenta que *num* se debe empezar con 1 y debe de aumentar independiente de los métodos, por ejemplo:

```
void met1(){  
  t1 ...  
  t2 ...  
}  
  
void met2(){  
  t3 ...  
  t4 ...  
}
```

#### 5.1.2 Etiquetas

Las etiquetas se crearán respetando la siguiente sintaxis:

*L<num>:*

tomando en cuenta que *num* se debe empezar con 1 y debe de aumentar independiente de los métodos, por ejemplo:

```

void met1(){
L1:
//instrucciones 3 direcciones
L2:
//instrucciones 3 direcciones
L3:
//instrucciones 3 direcciones
.
.
.
}

void met2(){
L4:
//instrucciones 3 direcciones
}

```

### 5.1.3 Operadores aritméticos

Las operaciones aritméticas que se realizarán serán las operaciones más básicas:

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Módulo	%

### 5.1.4 Operadores relacionales

Los operadores relacionales que se realizarán serán las operaciones más básicas:

Operación	Símbolo
Comparación	==
Diferencia	!=
Mayor	>
Mayor o igual	>=
Menor	<
Menor o igual	<=
Verdadero	1==1
Falso	1==0

**Nota:** los operadores lógicos del lenguaje de alto nivel (ver sección 4 de este documento) se deben de realizar por medio de cortocircuito, como se ha visto en clase y laboratorio.

### 5.1.5 Proposición de asignación

Las proposiciones de asignación tendrán las siguientes formas:

donde *op* puede ser un operador aritmético o relacional.

*identificador1 = identificador2 op identificador3;*

el valor de *identificador2* se le asignará a *identificador1*

*identificador1 = identificador2;*

asignación de un valor constante a un identificador

*identificador =valor;*

Ejemplo:

```
void met1(){  
t1 = 2;  
t2 = 5;  
t3 = t1 + t2;  
t4 = 10;  
t5 = t3 * t4;  
}
```

### 5.1.6 Salto incondicional

El salto incondicional no depende de una condicionante para realizar dicho salto de etiqueta, su sintaxis es la siguiente:

*goto Etq;*

Ejemplo:

```
void met1(){  
L1:  
t1 = 2;  
t2 = 5;  
goto L2;  
t3 = t1 + t2;  
t4 = 10;  
t5 = t3 * t4;  
L2:  
//instrucciones 3 direcciones  
}
```

### 5.1.7 Salto condicional

El salto condicional depende directamente de una condicionante para realizar dicho salto de etiqueta, su sintaxis es la siguiente:

*if (identificador1 oprel identificador2 ) goto Etq;*

dónde *oprel* representa un operador relacional y *Etq* representa una etiqueta.

Ejemplo:

```
void met1(){
L1:
t1 = 2;
t2 = 5;
if (t1 > t2) goto L2;
t3 = t1 + t2;
t4 = 10;
t5 = t3 * t4;
L2:
//instrucciones 3 direcciones
}
```

### 5.1.8 Declaración y llamada de métodos

En el código de tres direcciones un método será declarado de la siguiente forma:

```
void identificador(){} 
```

Ejemplo:

```
void metodo1(){
//instrucciones 3 direcciones
}
```

Para las llamadas a métodos se utilizará la siguiente sintaxis:

```
identificador();
```

Ejemplo:

```
void metodo1(){
metodo2();
}

void metodo2(){
//instrucciones 3 direcciones
}
```

### 5.1.9 Acceso a pila (Stack)

La pila (stack) estará representado por un arreglo y la manera de acceder a este será la siguiente:

Para asignar un valor al stack

*stack[identificador1] = identificador2;*

Para obtener un valor del stack

*identificador2 = stack[identificador];*

### 5.1.10 Acceso a memoria (Heap)

La memoria(heap) estará representado por un arreglo y la manera de acceder a este será la siguiente:

Para asignar un valor al heap

*heap[identificador1] = identificador2;*

Para obtener un valor del heap

*identificador2 = heap[identificador];*

Ejemplo de stack y heap:

```
void calculadora__Mult(){
t1=p+0;
t2=stack[t1];
t3=t2+0;
t4=heap[t3];
t5=p+2;
stack[t5]=t4;
t6=p+0;
t7=stack[t6];
t8=t7+1;
t9=heap[t8];
t10=p+2;
t11=stack[t10];
t12=t11*t9;
t13=p+1;
stack[t13]=t12;
}
```

## 5.2 Funciones propias del lenguaje

### Función printf

Esta función lo que realizará indicar una salida en pantalla y su sintaxis es la siguiente:

*printf(parametro, identificador);*

donde *parámetro* puede tomar los siguiente valores:

Parámetro	Acción
"%c"	Imprime el valor carácter del identificador.
"%d"	Imprime el valor entero del identificador.
"%f"	Imprime el valor con punto flotante del identificador.

Ejemplo:

```
void metodo1(){  
    t1 = 65;  
    printf("%c", t1);  
}
```



## 6. Optimización de código de 3 direcciones

La optimización de código consiste en mejorar y valga la redundancia optimizar código de manera que el programa compilado mejore su rendimiento y sea más eficiente. La optimización de código se debe realizar utilizando el método de optimización por bloques.

### ● Optimización por Bloques

Se utilizarán tres casos para optimizar el código: subexpresiones comunes, propagación de copias y código inactivo, a continuación se describe cada uno de ellos:

#### ○ Subexpresiones comunes

Cuando se realizan cálculos aritméticos dentro del código intermedio encontraremos casos en los que una misma operación se calcula varias veces por lo que si los valores de los operandos no cambian entre la primera y segunda vez que calculamos la operación podemos modificar el código de tal manera que calculemos una sola vez la operación asegurando que dentro del mismo bloque los valores se conserven. Podemos optimizar dos tipos de subexpresiones:

##### ■ Locales

Buscando dentro de un mismo bloque todas aquellas subexpresiones comunes, tomamos la variable a la cual se asigna la subexpresión común, eliminamos las instrucciones en las cuales se utilizaba la subexpresión y actualizamos las referencias de las variables eliminadas.

##### ■ Globales

El caso de subexpresiones comunes globales siguen los mismos principios que las locales, solo que en un nivel general de todos los bloques que se encuentran en el grafo.

- Propagación de copias

Dentro del código generado encontraremos instrucciones de la forma  $u = v$  es decir, asignación directa de dos variables. Lo que dicta la propagación de copias es sustituir las referencias de  $u$  por  $v$  en el código que sigue a la instrucción.

- Código inactivo

Cuando se termina la verificación de la propagación de copias muchas veces quedan proposiciones que son calculadas pero nunca utilizadas, para estos casos la optimización que corresponde es eliminar las instrucciones que no son utilizadas siempre que no perjudiquen el sentido lógico del código generado.

## 7. Instrucciones de código ensamblador

Una vez generado el código intermedio establecido según los criterios vistos anteriormente, el siguiente paso consistirá en transformarlo en código assembler, con la finalidad de generar un archivo objeto (.o) por medio de la herramienta NASM y finalmente el archivo ejecutable con el compilador de GCC

### 7.1 Registros a utilizar

Para garantizar la compatibilidad de la arquitectura se deberá generar assembler bajo la arquitectura i386 de 32 bits. Se podrá utilizar registros de 32bits o menores

32bit	16bits	8bits	8bits
eax	ax	ah	al
ebx	bx	bh	bl
ecx	cx	ch	cl
edx	dx	dh	dl

## 7.2 Traducción de declaraciones

En assembler al igual que otros lenguajes de programación se puede declarar variables y arreglos de forma global, estas variables y arreglos globales representará a las variables temporales, a la pila y montículo del código de tres direcciones de la siguiente forma:

```
section .bss
    Stack: resd 10000 ;array tamaño 1000
    Heap: resd 10000 ;array tamaño 1000
    temp1: resd 1      ;array tamaño 1
    temp2: resd 1
    temp3: resd 1
    temp4: resd 1

section .data
    p: dd 0 ;apuntador numérica iniciada en 0
    h: dd 0
```

## 7.3 Traducción de procedimientos

Los procedimientos en código intermedio tienen su propia forma de ser traducidos al lenguaje ensamblador de nasm. A continuación se describe cómo realizar la misma.

### 7.3.1 Metodo main

En nasm es de importancia instanciar y describir el método main, ya que sin este método nasm no generará el archivo objeto (.o) la sintaxis es la siguiente:

```
segment .text
    global _main
_main:
    ;Instrucciones en assembler
    ret
```

### 7.3.2 Procedimientos y etiquetas

Los procedimientos en assembler son bloques de código que ejecutan instrucciones, también puede haber otro bloque de código dentro de un procedimiento a esto se le llama etiquetas.

### 7.3.2.1 Procedimientos comunes

Una Procedimiento en assembler contiene instrucciones que se ejecutan línea por línea la característica principal es que son llamados mediante la instrucción Call y terminan con un "ret" (la instrucción ret indica la finalización del procedimiento), ejemplo de uso de procedimiento:

```
segment .text
    global _main
_main:
    call Procedimiento;Llamada al procedimiento
    ret

Procedimiento:
    ;Instrucciones en assembler
    ret;
```

### 7.3.2.2 Etiquetas

Las etiquetas son muy similares a los procedimientos, la diferencia es que las etiquetas se ejecutan dentro de un procedimiento y son llamados mediante la instrucción jmp (salto). Ejemplo de uso de etiquetas:

```
segment .text
    global _main
_main:
    call Procedimiento ;Llamada al procedimiento
    ret

Procedimiento:
    ;instrucciones en assembler
    jmp L1
    ;instrucciones en assembler
L1: ;Declaración de etiqueta
    ;instrucciones en assembler
    ret ;finalización de la etiqueta L1
    ;instrucciones en assembler
    ret;
```

### 7.3.2.3 Procedimientos nativos

Nasm puede ejecutar procedimientos nativos, estos procedimientos incluyen interrupciones y/o llamadas al sistema. ejemplo del uso de procedimientos nativos.

```
segment .text
    global _main
    extern _ExitProcess@4; procedimiento nativo
_main:
    push 0
    call _ExitProcess@4; termina la aplicación
    ret
```

## 7.3.3 Traducción de expresiones comunes

Las expresiones comunes, tales como operaciones aritméticas, saltos, accesos a arreglos, entre otras, se describen a continuación.

Ejemplos de traducción de operaciones aritmética

Código 3 direcciones	Código assembler
temp2 = temp1 + 4;	mov eax, temp1 add eax, 4 mov temp2, eax
temp2 = temp1 * 8;	mov eax, temp1 mul 8 mov temp2, eax
temp2 = temp1 / 10;	mov eax, temp1 div 10 mov temp2, eax

## Ejemplos de traducción de acceso y asignación a arreglos

Código 3 direcciones	Código assembler
Stack[temp1] = temp2;	mov ebx, Stack mov eax, temp1 mov ecx, temp2 mov [ebx + eax],ecx
temp2 = Stack[temp1];	mov ebx, Stack mov eax, temp1 mov ecx, [ebx + eax] mov temp2, ecx

## 8. Manejo de errores

La aplicación debe de ser capaz de reportar los errores encontrados en el proceso de transformar el código de alto nivel (PHP) a código 3 direcciones, para cada error se debe de indicar la línea, columna, tipo de error y una descripción donde incluye el lexema que causa el error, los tipos de errores son los siguientes:

- Errores léxicos
- Errores sintácticos
- Errores semánticos

El reporte de errores deberá tener un formato html y será mostrado en la interfaz del cliente FileVersion de la siguiente manera:

Figura 17. Ejemplo de interfaz sugerida para el reporte de errores

Tabla de símbolos Resultados Errores			
▼ Línea	▼ Columna	▼ Tipo Error	▼ Descripción
1	2	Error sintáctico	Se encontró "php?" se esperaba "?php"
10	1	Error Semántico	Variable "\$VarNueva" no definida

## 9. Entregables y restricciones

Para este proyecto se solicita que se cumpla con una serie de restricciones y requerimientos mínimos para asegurar que se ha desarrollado de acuerdo a las características descritas en las secciones anteriores, y de esta manera cumplir con funcionamiento de la aplicación que permita realizar una calificación objetiva del proyecto.

### 9.1 Restricciones

- El módulo cliente FileVersion debe de ser desarrollado en Visual C#.
- Los módulos servidor FileVersion y parser FileVersion deben ser desarrollados sobre el lenguaje de programación Java.
- Uso de JavaCC para la traducción a código de tres direcciones y a código ensamblador.
- Uso de JavaCC para la interpretación del código de tres direcciones y la optimización por bloques del código de tres direcciones.
- Para generar el código objeto (.o) del código assembler generado se deberá utilizar el compilador de nasm, y para generar el ejecutable (.exe) se deberá utilizar el compilador de mingw/GCC.
- Uso de Apache Thrift para la comunicación entre módulos
- El proyecto se desarrolla de manera individual.
- Copias de código fuente o de gramáticas serán merecedoras de una nota de 0 puntos y los responsables serán reportados al catedrático de la sección, así como a la Escuela de Ciencias y Sistemas.

### 9.2 Requerimientos mínimos

Los requerimientos mínimos son funcionalidades que garantizan una ejecución básica del sistema, para tener derecho de calificación se debe de cumplir con los siguiente requerimientos:

- Cliente FileVersion
  - Interfaz gráfica
    - Terminal para ingreso de comandos de instrucciones
    - Reportes (Tabla de símbolos, errores, resultados)
  - Comunicación con Servidor FileVersion
  - Comandos de instrucciones
    - Creación de repositorio (Source init).
    - Commit (Source commit).
    - Update (Source update).
    - Generar C3D (Release -3D).
    - Interpretar C3D (Prueba -3D).
  - Carpeta de la Copia de Trabajo

- Servidor FileVersion
  - Archivo de registro de repositorios creados y los commits realizados para cada repositorio.
  - Carpeta del Repositorio
  - Comunicación entre Cliente FileVersion y Parser FileVersion
- Parser FileVersion
  - Traducción a código de tres direcciones:
    - Operaciones aritméticas
    - Operadores relacionales
    - Operaciones lógicas
    - Traducción de clases
      - Instancia de objetos
      - Include
    - Traducción de métodos y funciones
      - Archivo main
      - Paso de parámetros por valor
      - Variables locales
      - Sentencia de retorno
    - Traducción de instrucciones de flujo de control
      - IF, IFELSE
      - SWITCH
      - WHILE
      - DOWHILE
      - FOR
    - Funciones propias del lenguaje
      - ECHO
  - Ejecución del código de tres direcciones
  - Comunicación con Servidor FileVersion

## 9.3 Entregables

- Código fuente de la aplicación
- Archivos de las gramáticas utilizadas en el Parser FileVersion, escritas en JavaCC.
- Manual técnico.
- Manual de usuario.

Fecha de entrega: Viernes 20 de mayo